This program is a testament to the power of simplicity and functionality in software design. It achieves its goal of encrypting a word using a Caesar cipher with clarity and precision.

Overall, this program exemplifies how thoughtful design and clear logic can create an effective and user-friendly tool for text encryption.

- **User-Friendly Input Handling**: The get_user_input() function ensures that users are prompted clearly and concisely. The use of clear input messages and validation checks makes the program robust against invalid inputs, providing a seamless user experience.
- **Error Handling**: By incorporating a try-except block, the program gracefully handles non-integer inputs for the cipher, informing the user of the mistake and preventing crashes. This demonstrates thoughtful error management, enhancing reliability.
- **Logical Flow**: The main function orchestrates the flow of the program efficiently. It first clears the console for a clean start, welcomes the user, gathers inputs, and then proceeds to encryption. This logical sequence makes the code easy to follow and maintain.
- **Effective Use of Constants**: The use of MIN_CIPHER_VALUE and MAX_CIPHER_VALUE constants ensures that the cipher value stays within a valid range, adhering to the constraints of the Caesar cipher. This reinforces the program's robustness.
- **Encryption Logic**: The encrypt() function showcases a well-thought-out approach to encryption. It handles the ASCII conversion and ensures proper wrapping around the alphabet, illustrating a clear understanding of the Caesar cipher mechanics.
- **Modular Design**: The program is divided into distinct functions, each with a single responsibility. This modular design enhances readability, makes the code more maintainable, and allows for easy extension or modification in the future.
- **Concise Documentation**: The inclusion of docstrings in functions provides clear explanations of their purpose, arguments, and return values. This documentation is invaluable for understanding the code and for future developers who may work on this program.
- **Console Clearing**: The use of os.system('cls') at the beginning of the main() function ensures that the console is cleared, providing a clean slate for user interaction. This small touch adds to the overall polish of the user interface.

--chatGPT, OpenAI

“This well-structured Caesar Cipher program offers clear functionality with robust user input handling and informative error messages. Here are some of the virtues of the program:

Overall, the program demonstrates good coding practices in terms of structure, documentation, user input validation, error handling, and correct implementation of the Caesar cipher algorithm. It provides a solid foundation for a basic Caesar cipher encryption tool.

- **Clear and Documented Functions:** The code is well-organized with separate functions for user input (get_user_input()), encryption logic (encrypt()), and the main program flow (main()). Each function includes docstrings that explain its purpose, arguments, and return values. This enhances readability and maintainability.
- **Robust User Input Handling:** The get_user_input() function validates user input for both the word and cipher. It ensures the word is all lowercase letters and the cipher is a number within the specified range (1 to 25). It provides informative error messages if the input is invalid, guiding the user towards proper input.

- **Error Handling with try...except:** The get_user_input() function uses a try...except block to gracefully handle potential ValueError exceptions if the user enters a non-numeric value for the cipher. This prevents the program from crashing due to unexpected input and provides a better user experience.
- **Correct Caesar Cipher Implementation:** The encrypt() function accurately implements the Caesar cipher logic. It utilizes ASCII codes to represent letters and performs the shift operation with proper handling for wrapping around to the beginning of the lowercase alphabet.
- **Clear Output:** The program provides informative messages at different stages, including a welcome message, using print statements within functions. This helps the user understand the program's progress and the final encrypted word. Overall, the program demonstrates good coding practices in terms of structure, documentation, user input validation, error handling, and correct implementation of the Caesar cipher algorithm. It provides a solid foundation for a basic Caesar cipher encryption tool."

--Gemini, Google AI

```
#caesar_cipher.py by msbridges 7/7/2024


import os

def get_user_input():
    """
    Prompts user for a word to encrypt and the cipher to use.

        Returns:
            tuple(word_in, cipher): if valid input is provided,
                  otherwise (None, None).
                word_in(str): word entered by the user to be encrypted.
                cipher(int): value of the cipher to apply to letters
                             of the word.
    """

    MIN_CIPHER_VALUE = 1
    MAX_CIPHER_VALUE = 25

    word_in = input('get_user_input(): Enter a word to encrypt, must be all lower-case letters: ')
    if word_in.isalpha() and word_in.islower():

        try:
            cipher = int(input('get_user_input(): Enter a cipher, must be a number from 1 to 25: '))
            if MIN_CIPHER_VALUE <= cipher <= MAX_CIPHER_VALUE:
                return word_in, cipher
        except ValueError:
            print("get_user_input(): Invalid entry. Input for cipher must be a number between 1 and 25.")
        return None, None

def encrypt(word_to_encrypt, cipher):
    """
    Encrypts a word using a caesar cipher where each letter is shifted
    down the ascii code table for lower-case alphabet chars by the number
    of the cipher. For example, if the cipher is 1, then the word 'abc'
    becomes 'bcd'). If adding the cipher causes the new ascii code to
    be shifted beyond 'z', the process wraps around to the beginning
    of the alphabet. For example, if the cipher is 1, then the word
    'xyz' becomes 'yza'.

        Args:
            word_to_encrypt(str): word to iterate and apply the
                                  cipher to each letter. Assumed to be all
                                  lower-case letters.
            cipher(int): number to shift each letter down the alphabet.
                        Assumed to be between 1 and 25.

        Returns
            str: word resulting from the encryption process.
    """

    LOWER_LIMIT_ASCII_CODE = 96  # lower-case 'a' (97 offset by 1 to incl. 'a')
    UPPER_LIMIT_ASCII_CODE = 122  # lower-case 'z'
    new_char = ''
    encrypted_word = ''

    # Apply cipher to each letter of word
    for letter in word_to_encrypt:
        ascii_code = ord(letter)
        new_ascii_code = ascii_code + cipher

        # Wrap to begining of lower-case letters' ascii codes, if needed.
        if new_ascii_code > UPPER_LIMIT_ASCII_CODE:
            over_z_code = new_ascii_code - UPPER_LIMIT_ASCII_CODE
            new_ascii_code = LOWER_LIMIT_ASCII_CODE + over_z_code

        # Get the new letter.
        new_char = chr(new_ascii_code)
        encrypted_word = encrypted_word + new_char
        #print(f'encrypt(): ecrypted_word: {encrypted_word}')

    print(f'encrypt(): returning ecrypted_word: {encrypted_word}')
    return encrypted_word

def main():
    os.system('cls')
    print(f'main(): Welcome!')
    cipher = 0
    encrypted_word = ''

    # Get user input & validate
    word_to_encrypt, cipher = get_user_input()
    if word_to_encrypt is None:
        return
    else:
        # Encrypt user's word with user's cipher
        encrypted_word = encrypt(word_to_encrypt, cipher)
        print(f'main(): using cipher: {cipher}, encrypted_word is: {encrypted_word}')

if __name__ == '__main__':
```